Efficient mapping of hardware tasks on reconfigurable computers using libraries of architecture variants

Miaoqing Huang, Vikram K. Narayana, *and* Tarek El-Ghazawi
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, The George Washington University
mqhuang@gwmail.gwu.edu, {vikram,tarek}@gwu.edu

Abstract

Scheduling and partitioning of task graphs on reconfigurable hardware needs to be carefully carried out in order to achieve the best possible performance. In this paper, we demonstrate that a significant improvement to the total execution time is possible by incorporating a library of hardware task implementations, which contains multiple architectural variants for each hardware task reflecting tradeoffs between the resources utilization and the task execution throughput. We develop a genetic algorithm based mapping approach, which considers both task graph and target platform, and present results for an N-body simulation application using estimated numbers for resource utilization for the constituent tasks and based on actual architectural constraints from different reconfigurable platforms. The results demonstrate improvements of up to 85.3% in the execution time, compared to choosing a fixed implementation variant for each task while keeping a reasonable searching time.

1. Introduction

On state-of-the-art high-performance reconfigurable computers (HPRC), the FPGA device is tightly coupled to the microprocessor through a high-speed interconnect and serves as a reconfigurable co-processor. To achieve the best possible performance for the parts of an application mapped to the reconfigurable device, the constituent task graph must be partitioned across multiple FPGA configurations and appropriately scheduled. Partitioning across configurations gives rise to overheads, due to reconfiguration of the FPGA, as well as transfer of intermediate data, which is required before and after reconfiguration. Our previous work addresses this problem by proposing the Reduced Data Movement Scheduling (RDMS) algorithm that strives to minimize the various overheads [1].

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

The RDMS algorithm achieves results close to the optimal solution, but it considers the availability of only a single implementation for each hardware task. With only a fixed implementation variant, there would be imbalances between the processing throughputs of interacting tasks. This provides an opportunity for optimization, if multiple implementation variants are available. This paper describes the methodology that may be adopted for task mapping, when multiple implementation variants are available for each task, in a common repository or hardware library*. In order to efficiently select the appropriate implementation for each task, a generic algorithm based approach is proposed. The assumptions used as part of this work are: (1) all architecture variants have the same input/output interfaces; (2) implementation of all tasks are pipelined, allowing concurrent execution of tasks within a configuration; (3) the data processed by each configuration is large, allowing the initial latency to be neglected; (4) FPGA supports only full configuration.

From assumptions 2 and 3, it follows that the processing time of a particular FPGA configuration simply equals the processing time of the slowest task in the configuration. The time taken for off-chip data transfer is computed based on the interconnect bandwidth and the data volume transfer to/from the task edges. Denoting T_R as the full reconfiguration time of FPGA, the execution time T_{hwe} for the entire computation consisting of n configurations is the sum of $n \cdot T_R$, the processing time of the slowest task in each configuration, and off-chip data transfer time for each configuration.

$*$. Inspired by the *Telescoping Languages* by Kennedy *et al.* [2]

2. Proposed Approach Using Architecture-Variant Hardware Library

Given a function-level DAG and a hardware library of the corresponding hardware modules for each task node, optimization techniques are needed to (1) select the proper implementation variant $F_{i,j}$ for each task F_i, which can be considered as a mapping between the hardware tasks and the available implementations, and (2) schedule the hardware tasks efficiently across multiple FPGA configurations $C_1, \ldots, C_k$, to maximize the performance. Step (2) can conceptually be carried out by using the RDMS algorithm [1]. However, both steps need to be carried out together, since the choice of the implementation variant depends on the selected set of tasks for each configuration C_k; conversely, obtaining the best schedule and the corresponding number of configurations, k, depends on the implementation variant chosen.

The most straightforward approach to obtaining the solution is by performing an exhaustive testing of all possible combinations of implementation variants, for each of the tasks in the DAG. However, the computation time required for such an approach is prohibitive. For example, if there are N tasks with each of them having J possible implementation variants to choose from, the search space consists of J^N possible combinations. Considering a 13-node task graph in which each task has 4 different implementations, an exhaustive search will take approximately 20.55 hours on a 2.8GHz Linux box. Clearly, a better method is required for exploring the various options.

2.1. Genetic Algorithm - Formulation

Genetic algorithms comprise a class of search methods inspired by biological genetic processes such as mutation, crossover, selection of the fittest, etc. General implementation strategies include the representation of the solution as a bit string, generally called a chromosome. Each chromosome consists of genes. In our case, we choose to have a gene for each of the N tasks in the task graph; each gene represents the choice of a particular implementation variant for the task. For example, if there are J possible implementation variants for each task, then every gene would have $\log_2 J$ bits. Correspondingly, a chromosome, which is basically one possible selection of variants for all tasks, constitutes of $N \log_2 J$ bits.

With this formulation of the solution space, the adopted genetic algorithm is shown in Algorithm 1. The fitness function is application specific. In our case, it basically gives a high score to individuals

Algorithm 1: Genetic Algorithm Pseudocode

```
    Input: Random initial population of $Q$ chromosomes
    Output: New population after genetic evolution
1.1 repeat
1.2     Evaluate fitness of each chromosome in population;
1.3     repeat
1.4         Select two chromosomes from the current population;
1.5         Crossover based on crossover rate, generate two offsprings;
1.6         Step through all the bits in the offsprings, flip them based
            on mutation rate;
1.7     until a new generation has been created ;
1.8 until $K$ generations have been evaluated ;
```

Table 1. The Characteristics of RC Platforms

	SGI RC100	SRC-6	Cray XD1
FPGA Device	XC4VLX200	XC2V6000	XC2VP50
Full Configuration Time (ms)	966	130	1,824
Interconnect Bandwidth (GB/s)	2.1	1.4	1.4

(chromosomes) that result in a low execution time, after obtaining the scheduling using the RDMS algorithm. If T_{min_hwe} denotes the minimum execution time exhibited in the current generation of chromosomes, the fitness score of individual chromosome is $1/(T_{hwe} - T_{min\_hwe})$ if $T_{hwe} \neq T_{min\_hwe}$. Otherwise, the score is set to a constant, i.e., 10,000.

Another version of the fitness function is also used, which happens to result in a faster convergence to the solution. This version basically uses $1/(T_{hwe}$ - $T_{thwe})$ as the fitness function for all chromosomes, where T_{thwe} is the so called target execution time and can be a dummy value.

With respect to the selection step in Line 1.4 of Algorithm 1, the chance of being selected is proportional to the chromosome fitness. Roulette wheel selection is a method we have adopted, which generally selects the fittest members to go through to the next generation, although it is not guaranteed.

3. Results

The target application we intend to implement is a part of the astrophysical N-Body simulation, in which the so called gas dynamical effects are simulated using a smoothed particle hydrodynamics (SPH) method, with the corresponding task graph as shown in Figure 1, which consists of a total of 18 tasks. For the detailed description about the equations used in the SPH pressure force calculation, please refer to [1].

3.1. Testbed and Hardware Library Setup

We emulate the SPH pressure force calculation on three different RC platforms as shown in Table 1. In this work, we assume that each module in the

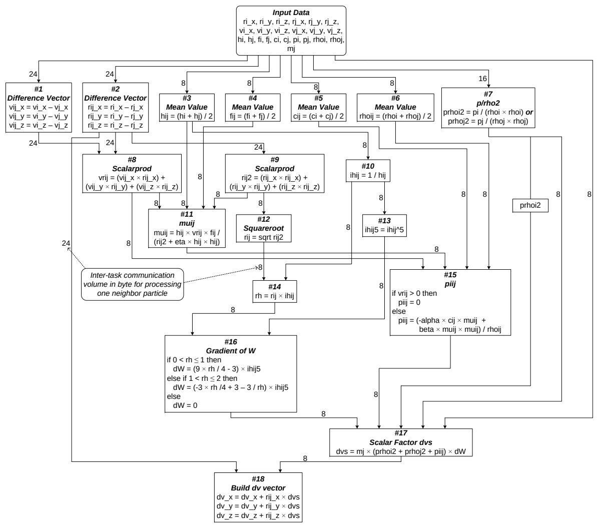

Input Data
ri_x, ri_y, ri_z, rj_x, rj_y, rj_z, vi_x, vi_y, vi_z, vj_x, vj_y, vj_z, hi, hj, fi, fj, ci, cj, pi, pj, rhoi, rhoj, mj
#1 Difference Vector
vij_x = vi_x – vj_x
vij_y = vi_y – vj_y
vij_z = vi_z – vj_z
#2 Difference Vector
rij_x = ri_x – rj_x
rij_y = ri_y – rj_y
rij_z = ri_z – rj_z
#3 Mean Value
hij = (hi + hj) / 2
#4 Mean Value
fij = (fi + fj) / 2
#5 Mean Value
cij = (ci + cj) / 2
#6 Mean Value
rhoij = (rhoi + rhoj) / 2
#7 p/rho2
prhoi2 = pi / (rhoi × rhoi) or
prhoj2 = pj / (rhoj × rhoj)
#8 Scalarprod
vrij = (vij_x × rij_x) + (vij_y × rij_y) + (vij_z × rij_z)
#9 Scalarprod
rij2 = (rij_x × rij_x) + (rij_y × rij_y) + (rij_z × rij_z)
#10
ihij = 1 / hij
#11 muij
muij = hij × vrij × fij / (rij2 + eta × hij × hij)
#12 Squareroot
rij = sqrt rij2
#13
ihij5 = ihij^5
prhoi2
Inter-task communication volume in byte for processing one neighbor particle
#14
rh = rij × ihij
#15 piij
if vrij > 0 then
piij = 0
else
piij = (-alpha × cij × muij + beta × muij × muij) / rhoij
#16 Gradient of W
if 0 < rh ≤ 1 then
dW = (9 × rh / 4 - 3) × ihij5
else if 1 < rh ≤ 2 then
dW = (-3 × rh /4 + 3 – 3 / rh) × ihij5
else
dW = 0
#17 Scalar Factor dvs
dvs = mj × (prhoi2 + prhoj2 + piij) × dW
#18 Build dv vector
dv_x = dv_x + rij_x × dvs
dv_y = dv_y + rij_y × dvs
dv_z = dv_z + rij_z × dvs
24
24
8
8
8
8
16
24
24
24
8
8
8
8
8
8
8
8
8
8
8
8
8
8
24
8
8
8
8
8
8
8

Figure 1. Data Flow Graph of SPH Pressure Force Calculation (with Assigned Node Number in Each Box)

architecture-variants hardware library consists of four implementations, imp_1 to imp_4. Imp_1 is the high performance version, which also consumes the most logic resources, so called "fully pipelined version". Imp_2, imp_3 and imp_4 occupy 50%, 25% and 12.5% of resources as imp_1 does. The throughput of the four implementations of the same hardware module share the same pattern as resource utilization.

Primitive operators, e.g., adder/subtractor and multiplier, are used to construct the functionality of nodes. In general, multiple primitive operators are used to build a pipelined hardware node. For instance, the fully-pipelined implementation of node #11 needs 1 adder, 4 multipliers and 1 divider. The resource utilization of pipelined double-precision (64-bit) floating-point operators based on the available literature are listed as follows, $+/-$(1,640 slices), $\times$(2,085 slices), $\div$(4,173 slices), $\sqrt{\ }$(2,700 slices). Table 2 lists the resource utilization of fully-pipelined version of each node composed by the primitive operators. The amount of slices occupied by each node is simply the summation of the slices of the primitive operators.

3.2. Experimental Setup and Results

In the implementation to emulate SPH pressure force calculation on three RC platforms, we assume all the calculations are carried out in double-precision (64-bit) floating-point format. As shown in Figure 1, the data of every particle consists of 13 scalar variables, i.e., 104 bytes. If we assume the number of particles to be emulated is $\mathcal{N}$ and the number of neighbors of each particle is 100, then the original storage requirement is $104\mathcal{N}$ bytes and the pipeline in Figure 1 needs to perform $100\mathcal{N}$ iterations. In the emulation carried out on three platforms, the number of particles is set to 16,000. Based on the input data volume and the throughput given in Table 2, the fastest implementation of all modules needs 16 ms to process the data. Correspondingly, the slowest implementation needs 128 ms to process the same amount of data.

The "task configuration time" (a fictitious value proportional to the task resource utilization) of the fully-pipelined task nodes and the inter-task communication time used for the RDMS algorithm are listed in Table 3 and Table 4 respectively.

While obtaining results using the proposed approach, the chromosome population of the genetic algorithm in Algorithm 1 is set to $Q = 100$; crossover and mutation rates are respectively taken to be 0.7 and 0.005. Since the number of implementation variants for each task is $J = 4$, two bits are used per gene, or 36 bits per chromosome.

Figures 2 show the progress of the estimated hardware execution time T_{min_hwe} during the iterations (or generations) of the genetic algorithm in the first 100

Table 2. Resource Utilization and Throughput of Fully-pipelined Task Nodes

Node No.	Operator* Combination	Slices	% of Device Utilization†			Throughput (GB/s)
			SGI	SRC-6	Cray	
1,2	3A	4,920	6.50	17.13	24.51	2.4
3,4,5,6	1A	1,640	2.17	5.71	8.17	0.8
7	1M,1D	6,258	8.26	21.79	31.17	1.6
8,9	2A,3M	9,535	12.59	33.20	47.50	4.8
10	1D	4,173	5.51	14.53	20.79	0.8
11,15	1A,4M,1D	14,153	18.69	49.27	70.50	3.2
12	1S	2,700	3.57	9.40	13.45	0.8
13	4M	8,340	11.01	29.04	41.55	0.8
14	1M	2,085	2.75	7.26	10.39	1.6
16	3A,4M,1D	17,433	23.02	60.69	86.84	1.6
17	2A,2M	7,450	9.84	25.94	37.11	3.2
18	3A,3M	11,175	14.76	38.91	55.67	3.2
Overall	24A,29M,5D,1S	123,390	162.94	429.59	614.68	

∗. A: adder/subtractor, M: multiplier, D: divider, S: square root.

†. Assume 15% of slices in device are reserved for vendor service logic.

Table 3. Configuration Time of Fully-pipelined Task Nodes on Three Platforms (ms)

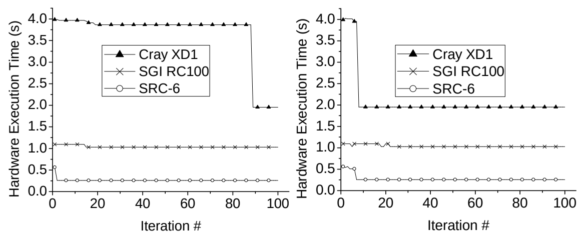
Node No.	Platform			Node No.	Platform		
	SGI	SRC-6	Cray XD1		SGI	SRC-6	Cray XD1
1,2	62.76	22.27	447.05	12	34.44	12.22	245.33
3,4,5,6	20.92	7.42	149.02	13	106.39	37.75	757.80
7	79.83	28.32	568.63	14	26.60	9.44	189.45
8,9	121.63	43.16	866.39	16	222.39	78.90	1584.03
10	53.23	18.89	379.17	17	95.04	33.72	676.94
11,15	180.55	64.06	1286.00	18	142.56	50.58	1015.40

Table 4. Inter-task Communication Time (ms)

Communicating-Node Pairs	Platform	
	SGI RC100	SRC/Cray
Category I*	6.10	9.14
Category II (0,7)	12.19	18.29
Category III†	18.29	27.43

*. (0,3),(0,4),(0,5),(0,6),(0,17),(3,11),(3,10),(4,11),(5,15),(6,15),(7,17),(8,15),(8,11),(9,11),(9,12),(10,14),(10,13),(11,15),(12,14),(13,16),(14,16),(15,17),(16,17),(17,18);

†. (0,1),(0,2),(1,8),(2,18),(2,8),(2,9).



(a) Without Target Value (b) With Target Value

Figure 2. Simulation of Hardware Execution Time

generations, for both versions of the fitness function. In both cases, the genetic algorithm manages to put all hardware tasks into one single configuration. This is primarily due to the large reconfiguration overhead, i.e., a reduction in the execution time by avoiding one extra configuration offsets the increase in execution time due to the choice of slower (and smaller) task variants. The algorithm will give more configurations if the reconfiguration time is relatively smaller to the task processing time, as demonstrated later in Section 3.3.

From Figure 2(b), it is observed that the searching on Cray XD1 platform converges more quickly if a target hardware execution time is given even if it is nowhere near the final target value.

Table 5 shows the hardware execution time for the cases when a fixed implementation variant is chosen for all tasks; a comparison with the genetic algorithm (GA) based approach shows that our approach results in a performance improvement by as much as 85.3%.

The genetic algorithm has been executed on the same Linux box used for the exhaustive approach. The time to finish the first 100 iterations of genetic algorithm is a little less than 30 seconds for all cases.

3.3. Use of more than one configuration

The use of a single configuration in previous section is mainly due to the large configuration time of the three platforms considered. If the configuration time is lower, or conversely, if the task processing time is larger compared to the reconfiguration time, the final mapping would have more configurations because reconfiguration no longer introduces a high overhead. To illustrate this fact, the developed mapping approach was used with larger task processing times for the SRC-6 case, termed as SRC-X. The hardware processing time of imp_1 to imp_4 are taken to be 50 ms, 100 ms, 200 ms and 800 ms respectively. As the results in Table 5 show, two FPGA configurations are used in this case, as against the single configuration used earlier.

Table 5. Comparison between Fixed Implementation and Genetic Algorithm

Platform	Number of Configurations					Hardware Execution Time (s)				
	imp_1	imp_2	imp_3	imp_4	GA	imp_1	imp_2	imp_3	imp_4	GA
SGI	2	1	1	1	1	2.098	0.998	1.030	1.094	1.030
SRC-6	5	3	2	1	1	1.059	0.724	0.516	0.258	0.258
Cray	7	4	2	1	1	13.264	7.698	3.959	1.952	1.952
SRC-X	5	3	2	1	2	1.229	0.928	0.788	0.930	0.688

4. Conclusion

In this paper, we have proposed a new methodology for hardware task mapping, based on the availability of multiple architectural variants for each hardware task. It is shown that the proposed approach significantly improves the the total execution time, by the use of tradeoffs in resource consumption and data throughput for each hardware task. In order to select the suitable task implementation variants in a reasonable time duration, a genetic algorithm approach is used. Results for the N-body simulation on three representative RC platforms show not only the effectiveness of the mapping approach, but also the efficiency of the process in finding the appropriate mapping.

Acknowledgment

The authors are grateful to Dr. Harald Simmler, Dr. Mohamed Bakhouya, and Olivier Serres for the help on RDMS algorithm and the genetic algorithm approach.

References

[1] M. Huang, H. Simmler, O. Serres, and T. El-Ghazawi, "RDMS: A hardware task scheduling algorithm for reconfigurable computing," in *Proc. the 16th Reconfigurable Architectures Workshop (RAW 2009)*, May 2009.

[2] K. Kennedy, B. Broom, A. Chauhan, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey, "Telescoping languages: a system for automatic generation of domain languages," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 387–408, Feb. 2005.